AlloyDB Omni for PostgreSQL - Analytical (OLAP) Benchmarking Guide

Sep 2023

Disclaimer

This AlloyDB Omni benchmark guide provides best practices for running On-Line Analytical Processing (OLAP) benchmarks. Your results may vary depending on several factors including, but not limited to the machine specifications of your AlloyDB Omni instance, type of client machine driving the benchmark, region, zone, and network bandwidth at the time of tests. Nothing in this user guide should be construed as a promise or guarantee about the results you'll derive from measuring the OLAP performance of AlloyDB Omni.

Overview

AlloyDB Omni is a downloadable edition of AlloyDB, designed to run anywhere — in your data center, on your laptop, at the edge, and in any cloud. AlloyDB Omni has several components and features, such as state-of-the-art log and transaction management, dynamic memory management, and a built-in columnar engine. As a whole, these features enable high performance for your transactional (OLTP), analytical (OLAP), and hybrid (HTAP) workloads.

The focus of this guide is to provide a step-by-step procedure to evaluate the analytical performance of AlloyDB Omni, which is powered by the `Columnar Engine` feature that stores and manages data in a columnar format. The Columnar Engine is designed and optimized for the efficient storage and retrieval of column data for analytical workloads where the emphasis is on efficiently processing large volumes of data compared to row-based data storage and to generate insights, analysis, and reporting. The analytical queries execute substantially faster because the Columnar Engine selectively accesses and processes only the columns of data that are pertinent to the query, resulting in significant query performance improvements. Users of AlloyDB Omni have a choice of running only transactional workloads (disable Columnar Engine) or run analytical queries along with transactional workloads (enable Columnar Engine and allocate appropriate memory).

Relational database systems typically require database administrators (DBAs) to optimize them for benchmarking, which includes configuring the transaction log settings, establishing the right buffer pool sizes, and tweaking other important database parameters (flags) and characteristics. These settings can also vary depending on the size and type of the instance. During installation, AlloyDB Omni chooses settings that are likely to be optimal for the number of CPUs and memory on your system. It requires minimal tuning of the Columnar Engine to achieve an optimal OLAP performance.

This document describes a step-by-step procedure to deploy and configure the AlloyDB cluster and a benchmark driving (client) machine, and provides steps to measure the performance of AlloyDB using a variety of OLAP benchmarks, including HammerDB TPROC-H (derived from TPC-H) with different scale factors and OLAP atomic queries developed internally at Google.

NOTE: Since HammerDB's TPROC-H implementation is a close variant of the official TPC-H benchmark, we will use the terms TPC-H and TPROC-H interchangeably throughout this user guide.

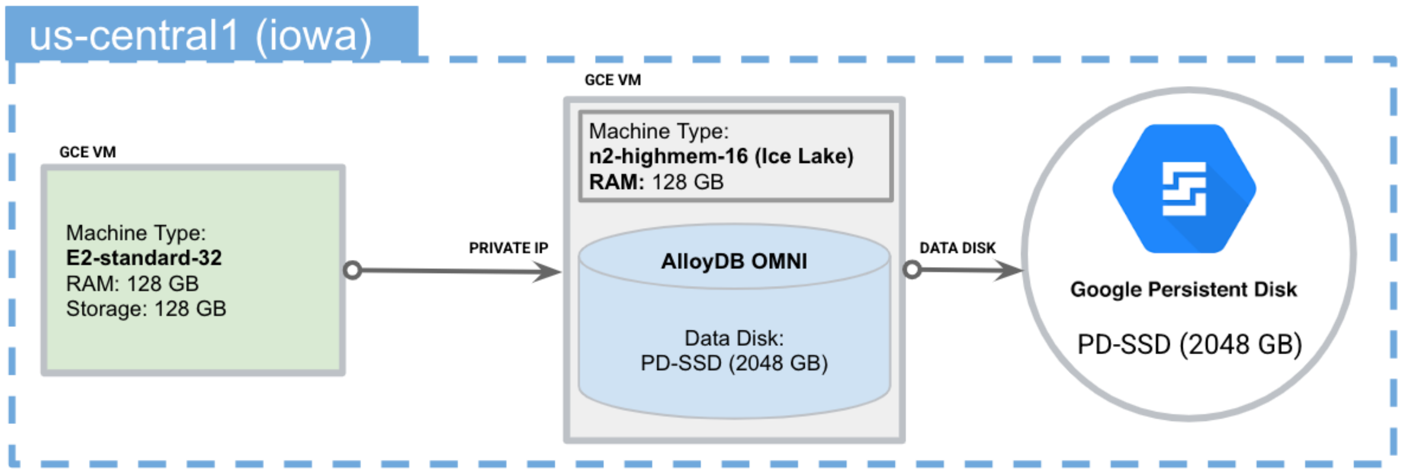
Benchmarking process

We'll go through the following steps to set up and run various OLAP benchmarks.

1. Configure AlloyDB Omni running on a Google Compute Engine (GCE) VM.
2. Setup of a separate benchmark driver client virtual machine running on GCE, where we will install benchmarking tools.
3. Install HammerDB on the client machine.
4. Run 2 benchmarks, "TPC-H like" and "OLAP atomics", using HammerDB.

Unless otherwise specified, we used the following setup for performance benchmarking:

Component	Value
AlloyDB Omni Machine Type	n2-highmem-16 16vCPU / 128GB / 2048GB Persistent Disk
AlloyDB Omni Version	`15.2.0` (This is the latest version at the time of writing of this document.)
Region	us-central1 (Iowa)
Zone	us-central1-a
Client VM — Machine Type	E2-standard-32 / 128GB / 128 GB persistent disk as boot disk NOTE: A large client machine can help you with faster load of TPC-H database. For power run of TPC-H, you don't need a large machine. Operating System: Debian Linux
Zone of Client VM	us-central1-a [same as AlloyDB Omni instance]
Connectivity	Private IP (same VPC)
Test tools	HammerDB-4.6 Psql
Workloads	TPC-H benchmark on a 16 vCPU machine with scale-factor of 10, 30 and 100 A collection of 11 targeted OLAP queries



In your own testing, you can run AlloyDB Omni on other platform configurations (as long as they meet these system requirements). Your benchmarking results will vary based on your specific hardware. Some crucial factors that can affect performance include the CPU model, number of cores/vCPUs, available memory, disk performance (IOPS and throughput), and network performance (latency and bandwidth) between the server and client.

Infrastructure Setup

Provision the server and client VMs

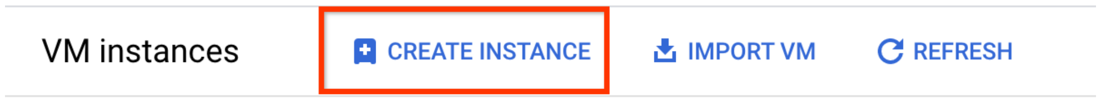
Note: The next section describes how to provision the VMs through the GCP cloud console. You may skip this section if running on your own hardware.

Provision Server on GCE

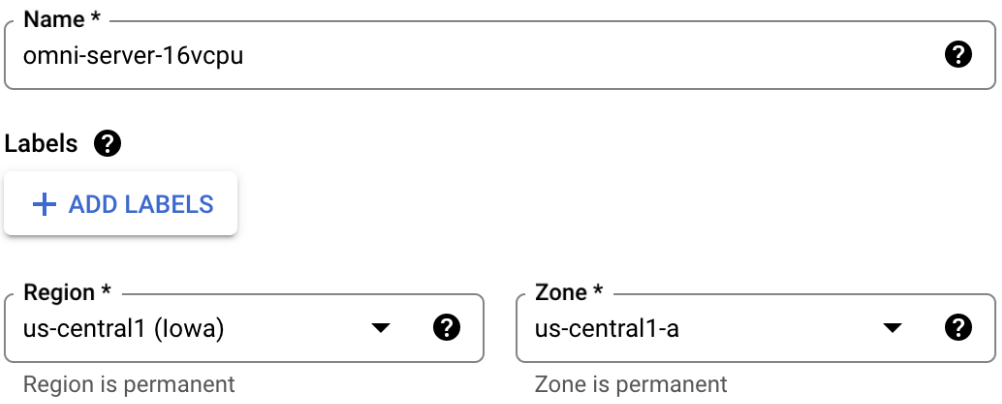
1. Create or select your GCP project: Go to https://console.cloud.google.com and select your project from the drop down menu or create a new one.

2. Follow these links on the portal: “Products and Solutions” → “All Products” → “Compute Engine”.

3. Click on the following button to create instance to run AlloyDB Omni.



4. Choose a name for your server VM, and select your desired region and zone.



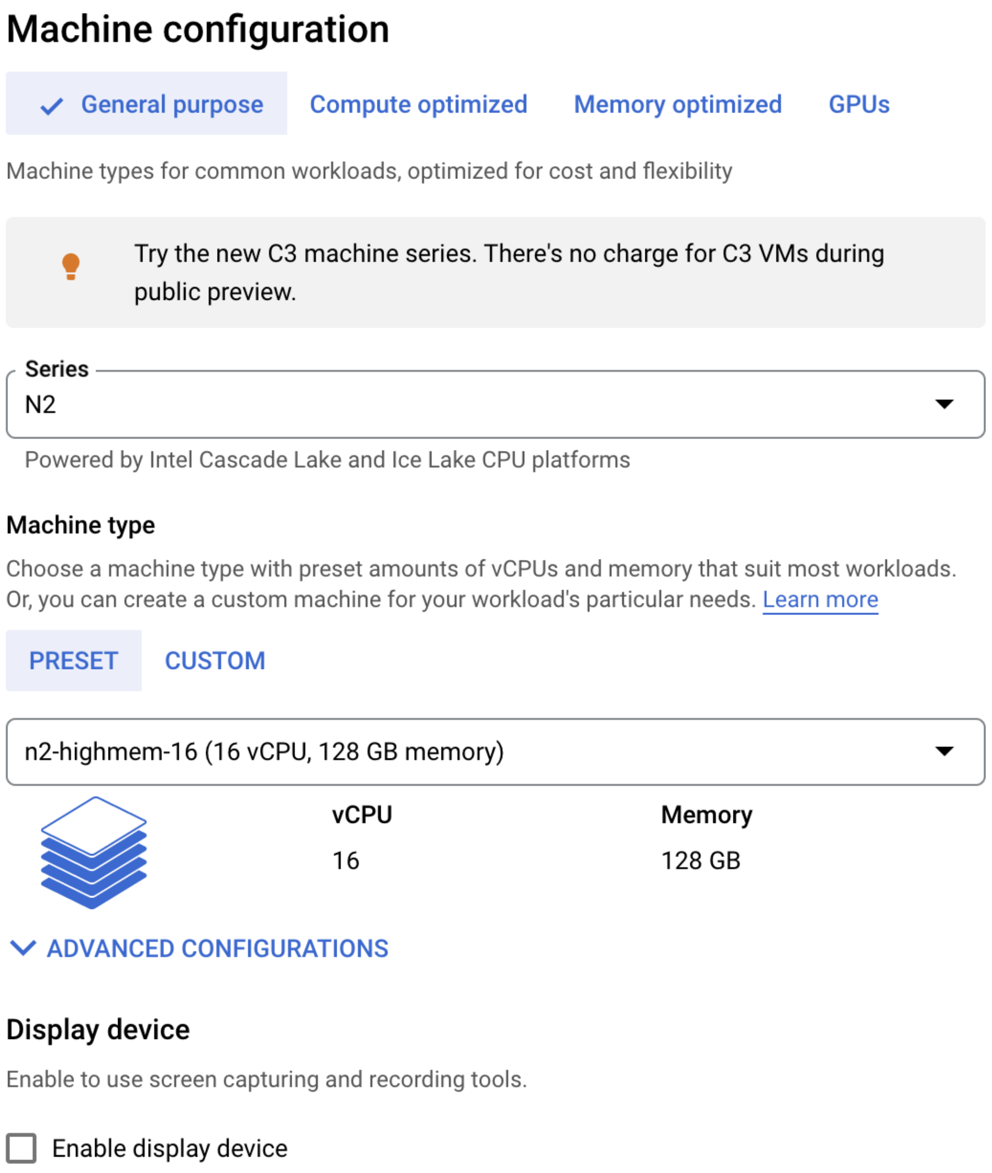
Name *
omni-server-16vcpu

Labels

Region *
us-central1 (Iowa)
Region is permanent

Zone *
us-central1-a
Zone is permanent

5. Under "Machine Configuration", select "N2" for "Series", and "n2-highmem-16" for the "Machine type".



Machine configuration

Machine types for common workloads, optimized for cost and flexibility

Try the new C3 machine series. There's no charge for C3 VMs during public preview.

Series: N2

Powered by Intel Cascade Lake and Ice Lake CPU platforms

Machine type

Choose a machine type with preset amounts of vCPUs and memory that suit most workloads. Or, you can create a custom machine for your workload's particular needs. Learn more

n2-highmem-16 (16 vCPU, 128 GB memory)

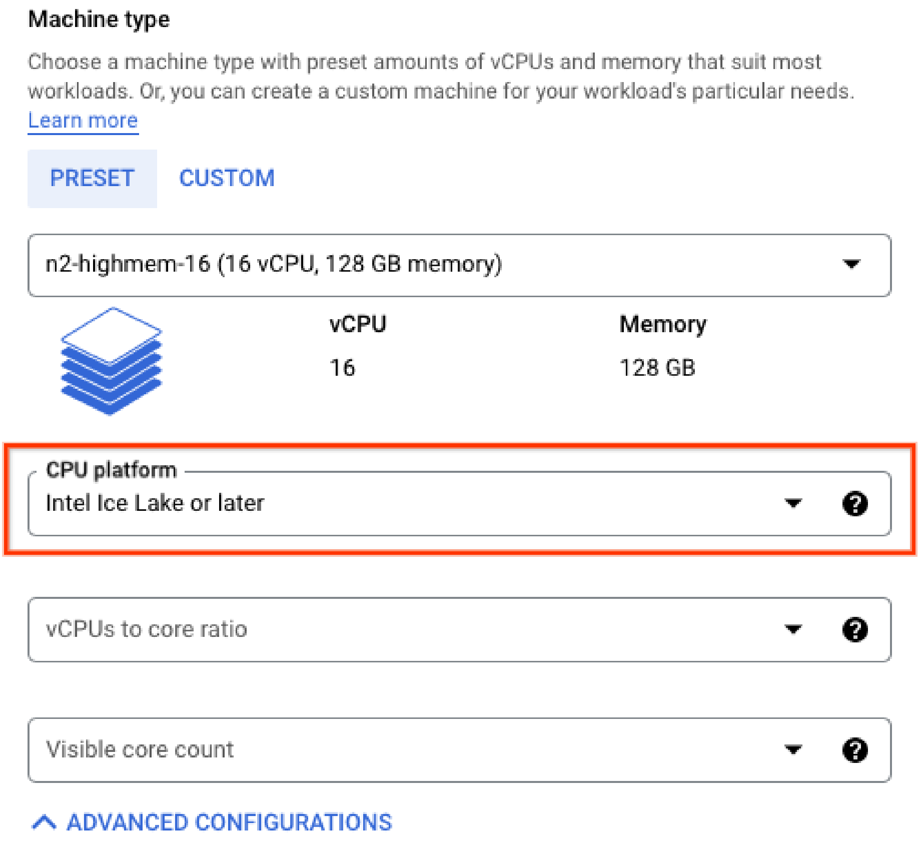
vCPU	Memory
16	128 GB

Display device

Enable to use screen capturing and recording tools.

☐ Enable display device

6. For best performance, expand "Advanced Configurations" and select "Intel Ice Lake or later".



Machine type

Choose a machine type with preset amounts of vCPUs and memory that suit most workloads. Or, you can create a custom machine for your workload's particular needs.
Learn more

n2-highmem-16 (16 vCPU, 128 GB memory)

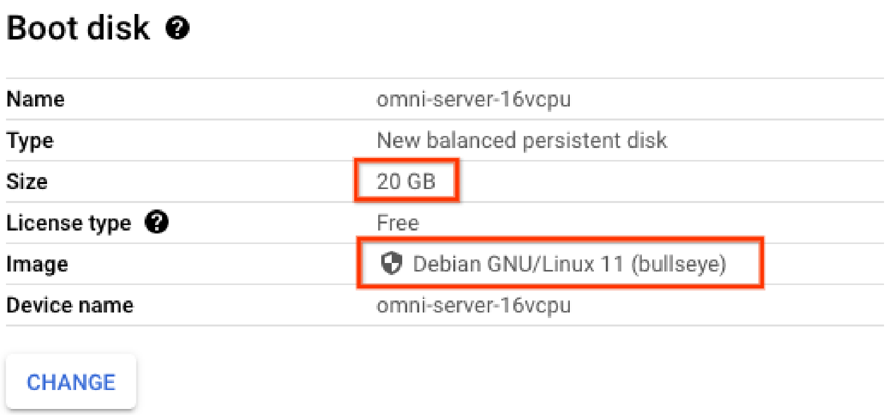
vCPU	Memory
16	128 GB

CPU platform: Intel Ice Lake or later

vCPUs to core ratio

Visible core count

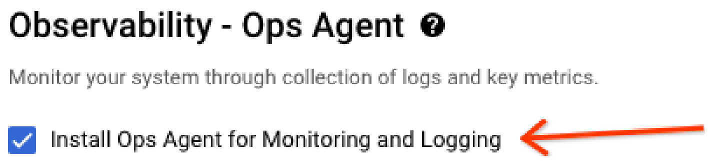
7. Under "Boot disk", ensure you are using a Debian 11 image, and have at least 20 GB provisioned for the boot disk.



Boot disk

Name	omni-server-16vcpu
Type	New balanced persistent disk
Size	20 GB
License type	Free
Image	Debian GNU/Linux 11 (bullseye)
Device name	omni-server-16vcpu

8. Under "Observability - Ops Agent", select "Install Ops Agent for Monitoring and Logging". This agent helps gather system metrics during the benchmark run.

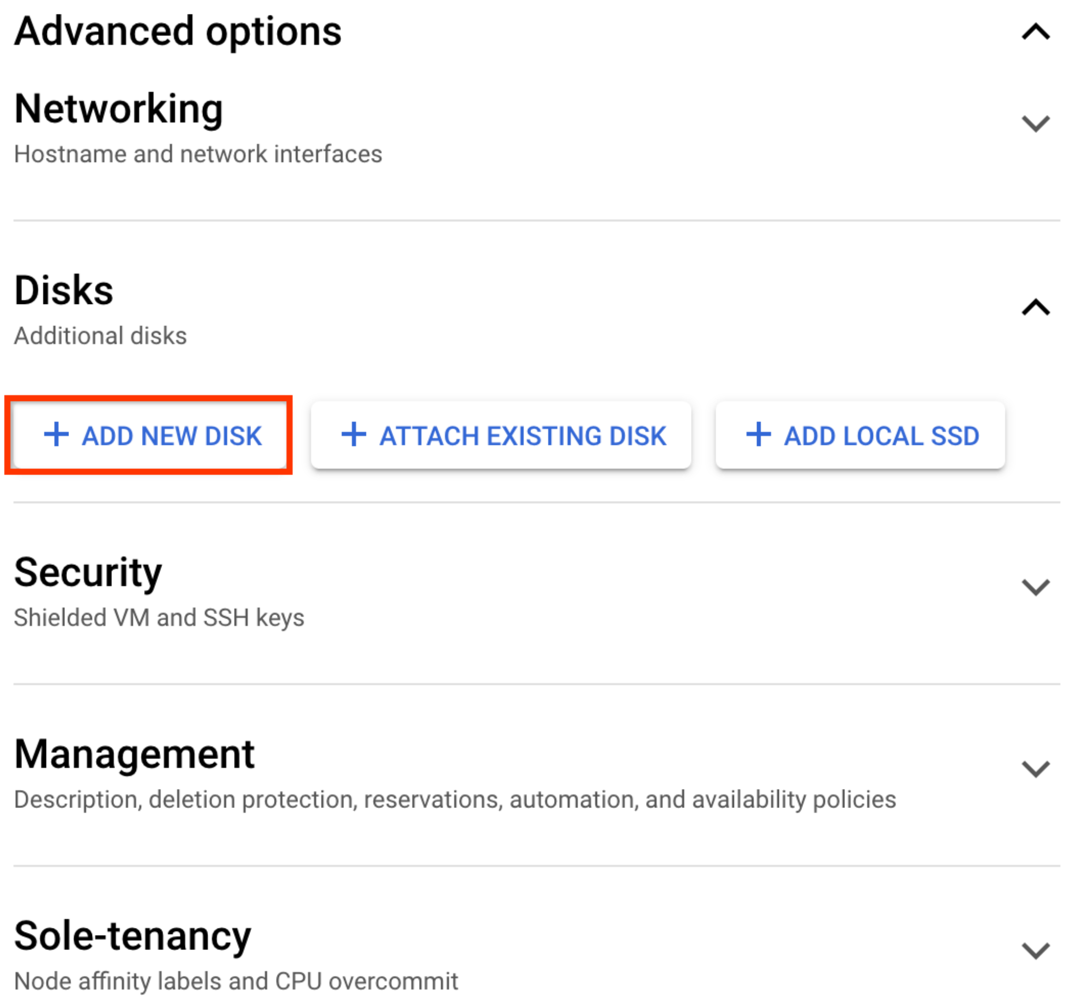


Observability - Ops Agent

Monitor your system through collection of logs and key metrics.

☑ Install Ops Agent for Monitoring and Logging

9. Next, we create a separate disk which will be used by the database. Under "Advanced options" → "Disks", select "Add new disk".


Advanced options

Networking
Hostname and network interfaces

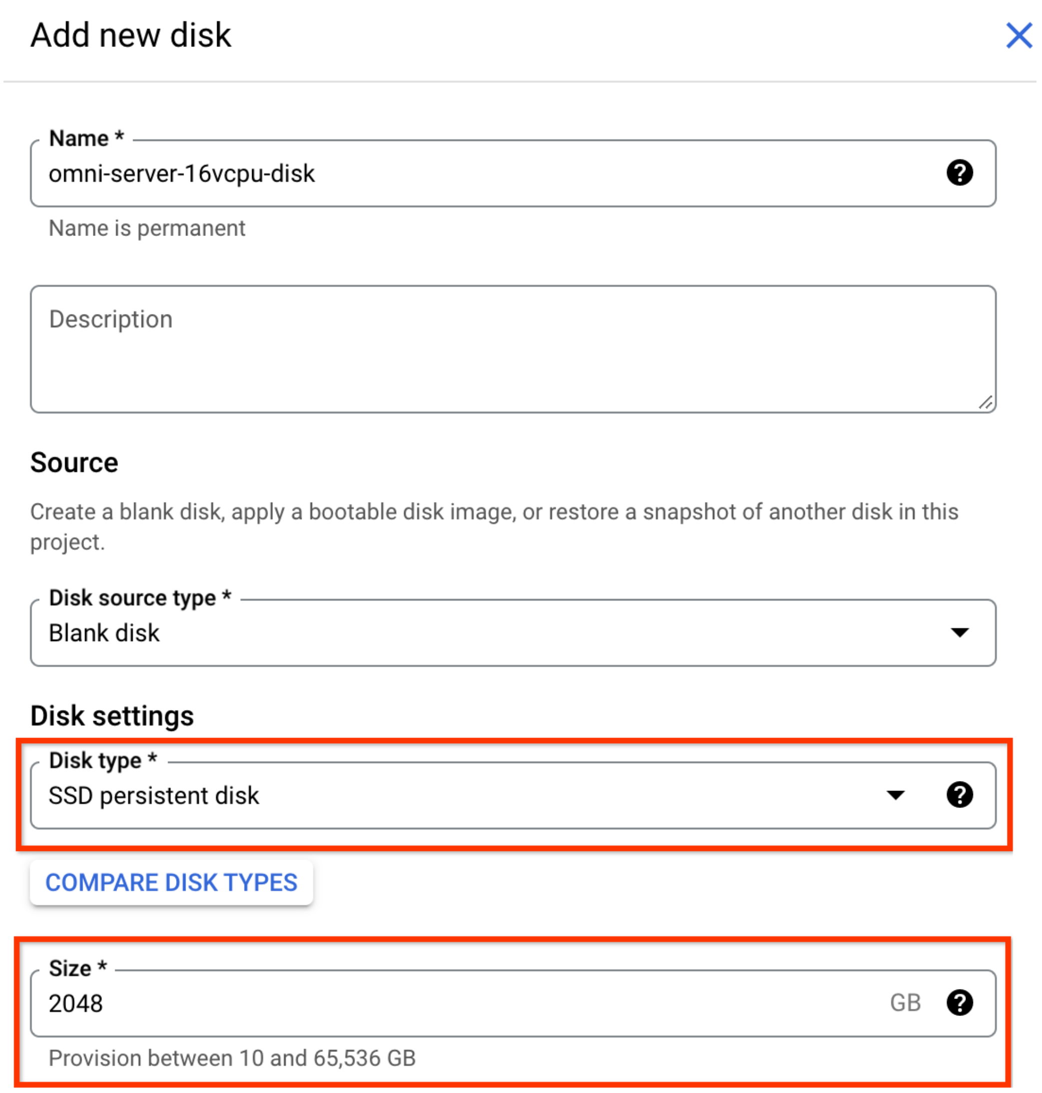
Disks
Additional disks

Security
Shielded VM and SSH keys

Management
Description, deletion protection, reservations, automation, and availability policies

Sole-tenancy
Node affinity labels and CPU overcommit

10. In the sidebar, ensure "Disk type" is set to "SSD persistent disk", and "Size" to "2048" GB. Persistent disks have per GB and per instance performance limits for the maximum IOPS and throughput that they can sustain, so we recommend a large disk size for better disk performance.



Add new disk

Name *
omni-server-16vcpu-disk
Name is permanent

Description

Source

Create a blank disk, apply a bootable disk image, or restore a snapshot of another disk in this project.

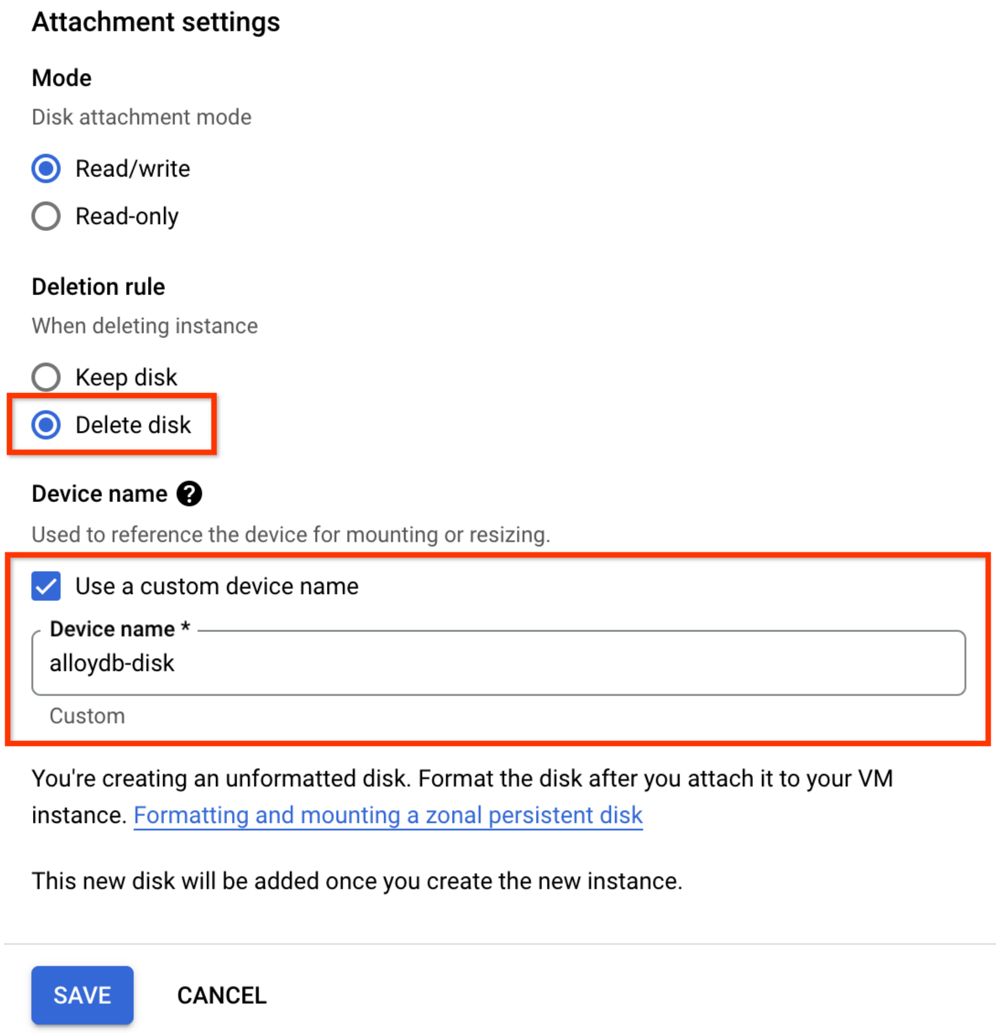
Disk source type *
Blank disk

Disk settings

Disk type *
SSD persistent disk

Size *
2048 GB
Provision between 10 and 65,536 GB

11. Finally, towards the bottom of the sidebar, ensure "Mode" = "Read/write", "Deletion rule" = "Delete disk", and use a custom device name "alloydb-disk". Then you may click "Save" to finish the disk setup.


Attachment settings

Mode

Disk attachment mode

● Read/write
○ Read-only

Deletion rule

When deleting instance

○ Keep disk
● Delete disk

Device name

Used to reference the device for mounting or resizing.

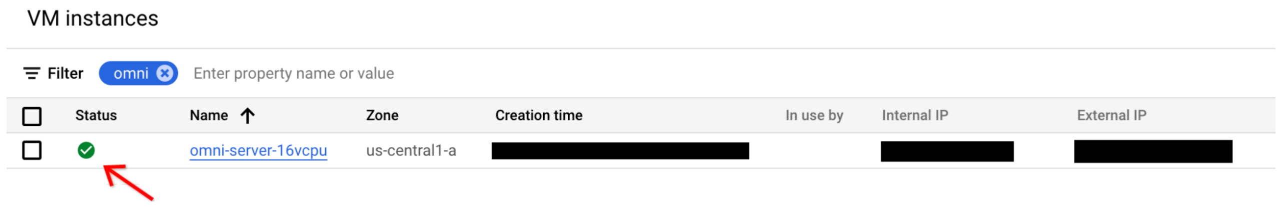
☑ Use a custom device name

Device name *
alloydb-disk
Custom

You're creating an unformatted disk. Format the disk after you attach it to your VM instance. Formatting and mounting a zonal persistent disk

This new disk will be added once you create the new instance.

12. Now click "Create" at the bottom of the create instance page, and a new VM will begin to be provisioned for you. Wait until the VM is fully created, which will be indicated by a green check mark under the "Status" column.


VM instances

	Status	Name	Zone	Creation time	In use by	Internal IP	External IP
☐	✓	omni-server-16vcpu	us-central1-a	[redacted]		[redacted]	[redacted]

Set up filesystem on server

Connect to the server VM using the **"gcloud compute ssh"** command. Refer this documentation for details "https://cloud.google.com/compute/docs/connect/standard-ssh".

Sample command:

```
gcloud compute ssh --zone "<primary zone>" "<server machine name>"  --project "<google-project>"
```

After connecting to the VM, run the following commands:

```
sudo mkdir -p /home/$USER/alloydb-data
sudo mkfs.ext4 -m 1 -F "/dev/disk/by-id/google-alloydb-disk"
sudo mount --make-shared -o noatime,discard,errors=panic "/dev/disk/by-id/google-alloydb-disk"
"/home/$USER/alloydb-data"
```

You can verify that you have formatted and mounted the PD correctly by running `lsblk -o NAME,MOUNTPOINT,FSTYPE,SIZE /dev/disk/by-id/google-alloydb-disk`:

```
$ lsblk -o NAME,MOUNTPOINT,FSTYPE,SIZE /dev/disk/by-id/google-alloydb-disk

NAME MOUNTPOINT              FSTYPE SIZE
sdb  /home/$USER/alloydb-data ext4     2T
```

Note the line that says `sdb /home/$USER/alloydb-data ext4 2T`: It means you have successfully formatted the PD with an ext4 filesystem, it is accessible through the path `/home/$USER/alloydb-data`, and it has 2TB capacity.

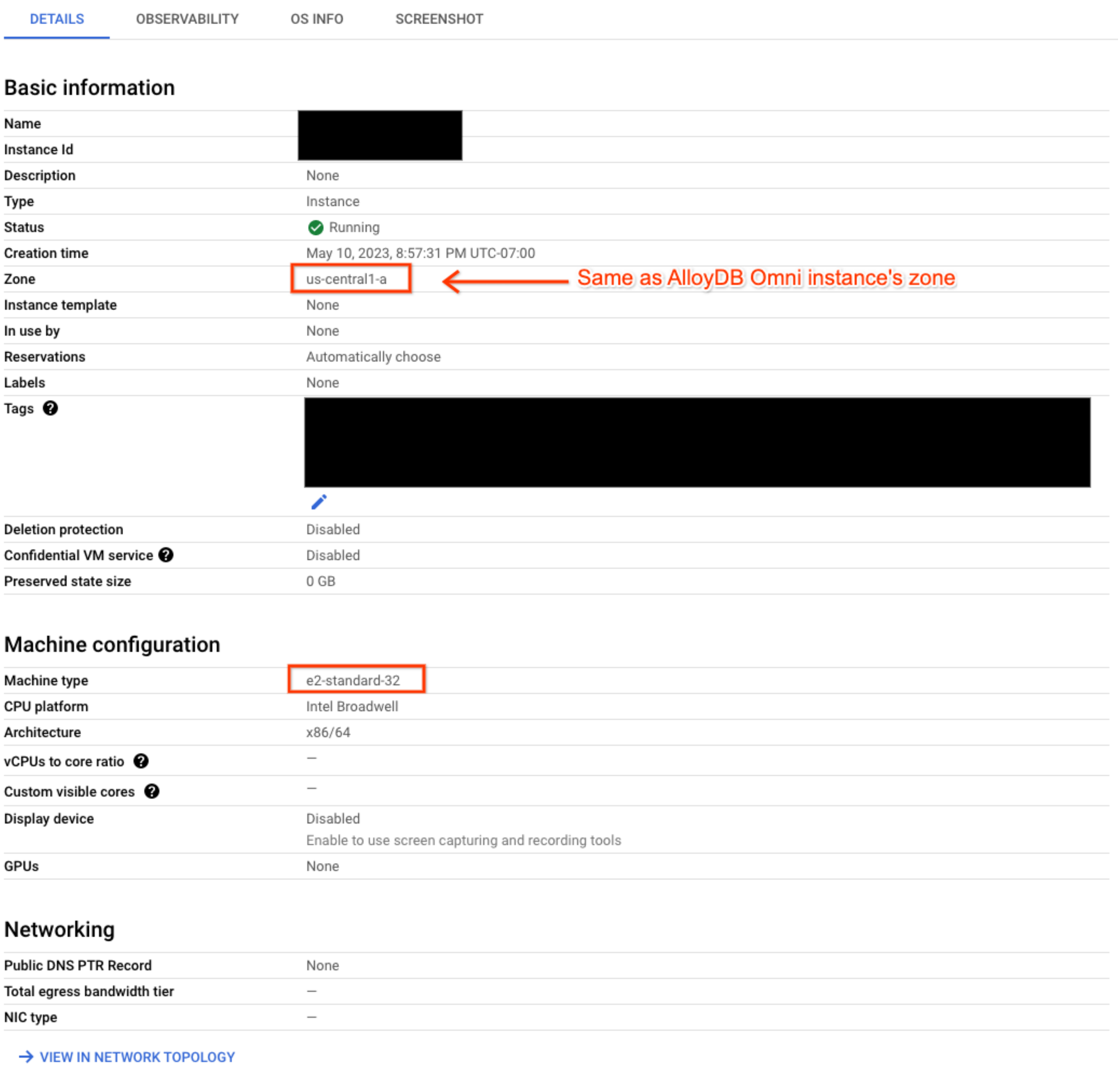
Provision Client on GCE

Unless otherwise specified, we used an E2-standard-32 VM (32 vCPUs, 128 GB memory) as a client for the TPC-H benchmarking. The client VM is created in the **same zone** as AlloyDB's primary instance.

For this analytical benchmarking guide, we will be primarily using TPC-H and OLAP atomic queries, and we do not need a large client VM to execute the benchmarks (i.e. queries). However, a large TPC-H database (especially, scale factor of size 30 or 100) will load faster with a larger client machine.

Important: For this exercise, the client must be provisioned in the same region, zone, and VPC as AlloyDB Omni's primary instance. Benchmarking tools directly access the AlloyDB instance over private IP. This setup reduces network latency between the server and client.

Below is a screenshot of client machine we provisioned for the purpose of this benchmarking guide.



Basic information

Name	[redacted]
Instance Id	
Description	None
Type	Instance
Status	Running
Creation time	May 10, 2023, 8:57:31 PM UTC-07:00
Zone	us-central1-a
Instance template	None
In use by	None
Reservations	Automatically choose
Labels	None
Tags	[redacted]
Deletion protection	Disabled
Confidential VM service	Disabled
Preserved state size	0 GB

Machine configuration

Machine type	e2-standard-32
CPU platform	Intel Broadwell
Architecture	x86/64
vCPUs to core ratio	—
Custom visible cores	—
Display device	Disabled Enable to use screen capturing and recording tools
GPUs	None

Networking

Public DNS PTR Record	None
Total egress bandwidth tier	—
NIC type	—

Install AlloyDB Omni

Follow the steps in "Install AlloyDB Omni on the VM" to install AlloyDB Omni.

Allow access from the client VM

Edit the file `/var/alloydb/config/pg_hba.conf`. This file controls which clients may connect to the database, and we need to add an entry for the client. For example, if your client's IP address is `1.2.3.4`, you will add a line at the end of `pg_hba.conf` like this:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
# "local" is for Unix domain socket connections only.
# Don't allow any unix socket connections as the alloydbadmin.
local   all             alloydbadmin                           reject
local   all             all                                     md5
# IPv4 local connections:
host    all             all             127.0.0.1/32            trust
# IPv6 local connections:
host    all             all             ::1/128                 trust
# Allow replication connections on localhost, from a user with the replication privilege.
local   replication     all                                     md5
host    replication     all             127.0.0.1/32            trust
host    replication     all             ::1/128                 trust
host    all             all             1.2.3.4/32              trust  # <-- ENTRY FOR CLIENT
```

NOTE: In this guide, we use the "trust" setting to simplify the benchmarking setup. However, note that "trust" bypasses password protection, and should not be used for a production instance.

Update database configuration

Finally, edit the file `/var/alloydb/config/postgresql.conf`. This file sets the configurations for the database. The default configuration values are already tuned for your system's vCPU and RAM, but in our benchmarking, we also set the following values, which are more tuned towards analytical queries:

```
work_mem=65536
default_statistics_target=200
google_columnar_engine.enabled=on
```

Start AlloyDB Omni

Now we may restart AlloyDB Omni to pick up the updated configurations:

```
sudo alloydb database-server stop
sudo alloydb database-server start
```

If you do not see any errors, that means AlloyDB Omni is running. Verify by connecting to the database locally:

```
sudo docker exec -it pg-service psql -h localhost -U postgres
```

You should see a psql prompt, which means you have successfully connected:

```
sudo docker exec -it pg-service psql -h localhost -U postgres
```

```
psql (15.2)
Type "help" for help.

postgres=#
```

Type `quit` to exit psql.

Setup of Benchmark Driver Machine (Client)

This section will guide you through the steps of configuring the client machine, where we will install benchmarking tools such as HammerDB.

Connect to the client machine using the **"gcloud compute ssh"** command.

Sample command:

```
gcloud compute ssh --zone "<primary zone>" "<client machine name>"  --project "<google-project>"
```

Install PostgreSQL client

You will need a `psql` client application to connect to AlloyDB Omni. Use the following command to install a postgresql client that includes a psql application and then ensure you are able to connect.

```
sudo apt-get update
sudo apt install -y postgresql-client
```

Now ensure that it works and you are able to connect to the AlloyDB Omni. Use the "Private IP" address of your AlloyDB Omni instance.

```
psql -h <Private IP> -U postgres
```

Install HammerDB-4.6 Driver for TPC-H benchmark

For this benchmarking guide, we utilized the HammerDB-4.6 driver. Execute the following commands to install HammerDB driver:

```
mkdir hammerdb
pushd hammerdb
curl -OL
https://github.com/TPC-Council/HammerDB/releases/download/v4.6/HammerDB-4.6-Linux.tar.gz
tar zxvf HammerDB-4.6-Linux.tar.gz
```

NOTE: If your operating system is Non-Debian, perform the following checks to ensure you have all essential libraries to run hammerdb. For debian, the steps below are optional.

```
cd HammerDB-4.6
sudo ./hammerdbcli
```

This puts you in the HammerDB shell. From there, run `librarycheck`:

```
HammerDB CLI v4.6
Copyright (C) 2003-2022 Steve Shaw
Type "help" for a list of commands
Initialized SQLite on-disk database /tmp/hammer.DB using existing tables (45,056 KB)
hammerdb> librarycheck
```

In the output, look for the section that says `Checking database library for PostgreSQL`, and ensure it succeeds. Otherwise, fix any errors that show up. On some platforms, you may need to install additional packages:

```
Debian-based system: sudo apt-get update && apt-get install -y libpgtcl
Red hat: sudo yum install tcl libpq
```

If the check was successful, you should see output like this:

```
~/hammerdb/HammerDB-4.6$ ./hammerdbcli
HammerDB CLI v4.6
Copyright (C) 2003-2022 Steve Shaw
Type "help" for a list of commands
Initialized new SQLite on-disk database /tmp/hammer.DB
hammerdb>librarycheck
<... snipped ...>
Checking database library for PostgreSQL
Success ... loaded library Pgtcl for PostgreSQL
<... snipped ...>
```

Notes on performance benchmarking

Benchmark Cleanup

This step is important if you are planning to execute multiple benchmarks in succession. Performing a proper cleanup between each benchmark is a critical prerequisite for accurate and reliable benchmarking results. This includes deleting previous benchmark data (i.e. benchmark database), and rebooting the AlloyDB Omni instance (that clears caches at database and operating systems level) before running another benchmark. A proper benchmark cleanup ensures that residual effects from previous benchmarks do not affect the performance measurements of the new benchmark. It also helps to ensure consistency and repeatability of the benchmark results, which is essential for making meaningful comparisons between different systems or identifying areas for optimization in hardware, software, or configuration.

Follow the URL https://cloud.google.com/compute/docs/instances/stop-start-instance to learn more about how to reboot a GCE VM.

To drop the previous benchmark database, you can use the following psql command from the client machine.

```
psql -h <Private-IP> -U postgres -c "DROP DATABASE IF EXISTS <database_name>;"
```

Understanding system performance

Since AlloyDB Omni can be run on many different environments, it is important to know that the transaction performance is highly dependent on CPU/Memory/IO/Network latency.

1. When most data fits in memory, it is a CPU bound workload, and more CPUs will get more transaction performance.
2. When most data can not fit in memory, it becomes an IO bound workload, and more disk IOPS/throughput will get better query performance.
3. Query latency is affected by network latency between client and server communication. It is recommended to have the client and server located in the same local network or same zone for benchmarking purposes.

Before benchmarking, it is useful to be able to characterize system performance of the hardware. In this section, we list down some commands that can be used to measure:

1. Performance of the CPU
2. Performance of the disk
3. Network latency between client and server

CPU performance

CPU performance can be measured by the sysbench benchmark, see https://github.com/akopytov/sysbench for installation instructions.

Use the following command to measure cpu performance:

```
sysbench cpu --cpu-max-prime=10000 --threads=<Number of vCPUs> run
```

Disk performance

Fio can be used to measure disk performance.

Use the following commands to measure IOPS, throughput and latency.
IOPS

```
fio --time_based --runtime=60s --ramp_time=2s --ioengine=libaio --direct=1 --name=iops_test
--filename=/mnt/disks/pgsql/fio_test --bs=8k --iodepth=256 --size=4G --readwrite=randrw
--rwmixread=25 --verify=0 --group_reporting=1
```

Write Throughput

```
fio --name=write_throughput --filename=/mnt/disks/pgsql/fio_test --numjobs=16 --size=4G
--time_based --runtime=60s --ramp_time=2s --ioengine=libaio --direct=1 --verify=0 --bs=256k
--iodepth=256 --rw=randwrite --group_reporting=1
```

Latency

```
fio --time_based --runtime=60s --ramp_time=2s --ioengine=libaio --direct=1 --name=latency_test
--filename=/mnt/disks/pgsql/fio_test --bs=256k --iodepth=1 --size=4G --readwrite=randwrite
--verify=0
```

Network latency

Ping can be used to measure network latency.

```
ping <IP address> -c 100
```

TPC-H Benchmark

HammerDB is a popular benchmarking tool that includes a TPC-H (a standard decision support benchmarking tool) implementation that we use for evaluating performance of OLAP support in AlloyDB PostgreSQL. HammerDB TPC-H measures the performance of a database system by executing a set of 22 standard queries. The TPC-H benchmark is a widely accepted industry standard benchmark for decision support systems that involves complex queries and large data sets.

This section provides a comprehensive guide on how users can customize HammerDB to execute the TPC-H benchmark to gauge the performance of the AlloyDB PostgreSQL database system.

Prerequisites

A. You need to run the following steps from a client (driver) machine. Ensure that you have completed the setup steps listed in the "Setup of Benchmark Driver Machine (Client)" section (especially installation of the HammerDB utility).

B. **Cleanup**: If you are running multiple benchmarks in succession, ensure you follow the "Benchmark Cleanup" section before doing your subsequent run.

Initial Setup on Client Machine

Connect to the client machine and execute all the following commands from the `hammerdb/HammerDB-4.6` directory.

```
cd hammerdb/HammerDB-4.6
```

Then create `setup.env` file as follows:

```
cat << EOF > setup.env

# Private IP of the AlloyDB primary instance
export PGHOST=111.222.333.444

# Postgres default port address. You do not need to change it unless you use non-default port
address.
export PGPORT=5432   # default port to connect with postgres

# TPC-H Scale Factor (determines the size of the database that we want to build).
export TPCH_SCALE=10

EOF
```

Edit the generated `setup.env` file and change the above parameter values to those that are suitable to your environment setup.

For the purpose of this benchmarking guide, we evaluate the performance using three important scale factor (`TPCH_SCALE`) sizes (i.e. 10, 30 and 100) of the TPC-H benchmark.

In the context of TPC-H benchmark, scale factor refers to the size of the data set used in the benchmarking process. The scale factor determines the number of rows in the TPC-H database tables and it represents the volume of data to be processed by TPC-H queries.

The scale factors **10**, **30**, and **100** represent data sets of approximate sizes **20GB**, **60GB** and **200GB**, respectively. The significance of trying these different scale factors is to evaluate the performance of the database system under varying data volumes and workloads.

When a database system is tested with a smaller scale factor, such as 10, it may perform well as the data set size is relatively small. However, as the data set size increases, the performance of the database system may decrease due to increased resource consumption, buffer cache hit misses, and other processing overheads. Testing the database system with larger scale factors, such as 30 or 100, can help identify potential performance bottlenecks and scalability issues in the database system that may arise under heavy workloads and larger data sets.

Furthermore, testing with different scale factors helps to evaluate a database system's ability to scale with increasing data sizes. This information can be useful for organizations that need to handle large amounts of data and require a database system that can scale efficiently to meet their needs.

NOTE: The number of users (or clients) is set to 1, since this user guide is only running TPC-H in [power mode] and not throughput mode.

Script to load TPC-H data

For the TPC-H benchmark, a "**load step**" refers to the process of populating the benchmark database with initial data before running the actual performance test.

During this step, the benchmarking tool inserts data into the `tpch` database tables according to the specified scale factor. The purpose of the load step is to create a realistic workload for the performance test and to ensure that the test results are comparable across different systems.

After the load step is completed, the database is in a consistent state with a defined set of initial data, ready to be used for the TPC-H benchmark test.

Follow the steps below to load the TPC-H database:

1. Switch to the benchmark home directory.

```
cd hammerdb/HammerDB-4.6
source ./setup.env
```

2. Create **`build-tpch.sh`** file as follows:

```
#!/bin/bash -x
source ./setup.env

./hammerdbcli << EOF

# CONFIGURE PARAMETERS FOR TPC-H BENCHMARK
# --------------------------------------
dbset db pg
dbset bm tpc-h

# CONFIGURE POSTGRES HOST AND PORT
# --------------------------------------
diset connection pg_host $PGHOST
diset connection pg_port $PGPORT

# CONFIGURE TPC-H
# --------------------------------------
diset tpch pg_tpch_superuser postgres
diset tpch pg_tpch_user postgres
diset tpch pg_tpch_dbase tpch

# --------------------------------------
diset tpch pg_scale_fact $TPCH_SCALE
diset tpch pg_num_tpch_threads 32
diset tpch pg_refresh_on false

diset tpch pg_refresh_verbose false
diset tpch pg_degree_of_parallel 8
vuset vu 1

# logging
vuset logtotemp 1
vuset timestamps 0
vuset unique 0

# load and run benchmarking script
loadscript
buildschema

# terminate when completed
vudestroy
quit
EOF
```

3. Execute the load command as shown below and wait for the command to finish. During this command, you may view the contents of `results/build-tpch.out` in a second terminal to see its progress.

```
chmod +x ./build-tpch.sh
mkdir -p results
sudo nohup ./build-tpch.sh > results/build-tpch.out 2>&1
```

4. **Validate Load**: The load step is an important aspect of the TPC-H benchmark because it affects the benchmark's accuracy and repeatability. The quality and consistency of the data that is loaded into the database can have a significant impact on the performance measurements, and therefore, it is important to validate that the load step is executed properly.

   Use the following commands to validate the load quickly:

```
$ . ./setup.env
$ psql -h $PGHOST -U postgres
postgres=> \l+ tpch
                                                                             List of
databases
     Name     |      Owner        | Encoding | Collate |  Ctype  |           Access
privileges           |  Size    | Tablespace |                Description

--------------+------------------+----------+---------+---------+----------------------
-----------------+---------+------------+--------------------------------------------
 tpch         | postgres             | UTF8     | C.UTF-8 | C.UTF-8 |
                      | --- GB   | pg_default |
```

The scale factors 10, 30, and 100 represent data sets of **approximate** sizes **20GB**, **60GB** and **200GB**, respectively. Ensure that the size of the `tpch` database matches the scale factor of your choice.

Columnar Engine (CE) Flags

AlloyDB's Columnar engine related parameters (flags) come with optimal settings and no tuning is generally required except that the columnar engine is to be enabled. However, for this user guide, updating them with proper values allows for efficient processing of analytical queries, reduces query response times, and improves resource utilization, which are critical factors for organizations that need to handle large volumes of data and require fast and accurate analysis of that data.

Important Flags to Tune

The following are the database flags that we tune to enhance the efficacy of OLAP workloads:

Database Flag	Is AlloyDB Unique?	Description	Default Value
work_mem	No	Increasing the work_mem value can improve the performance of queries that perform a lot of temporary work (like sorting, hashing, bitmap, *etc.*). If your AlloyDB instance does not have adequate memory, a very high value of work_mem may cause performance issues.	16MB
default_statistics_target	No	Increasing the default_statistics value can improve the accuracy of the query planner's estimates, which can lead to better performance for queries that access the column. However, significantly high values can also increase the time it takes to analyze the table.	100
google_columnar_engine .enabled	Yes	This configuration flag in AlloyDB specifies whether the Columnar Engine is enabled or not. The Columnar Engine is a new feature in AlloyDB that can significantly improve the performance of analytical queries.	OFF
google_columnar_engine .memory_size_in_mb	Yes	This flag in AlloyDB specifies the amount of memory that is allocated to the columnar engine. The default value is ~30% of the RAM on the VM, but it can be increased or decreased depending on the needs of your database.	~30% of the RAM size
google_columnar_engine .relations	Yes	This configuration flag in AlloyDB specifies a set of tables and their columns that need to be stored in columnar format. The columnar format is a more efficient way to store data for analytical queries, so using this flag can improve the performance of those queries.	Empty string.

Tuning for Scale Factors 10 and 30

Since the database sizes for scale factors 10 and 30 are significantly smaller than available RAM (128GB) on the 16 vCPU machine type, we can simply allow all the entire `tpch` database (i.e. all the columns of all `tpch` relations) to be populated in the columnar engine.

1. Connect to the **client** machine and run the following command to edit important Columnar Engine flags:

```
source ./setup.env
psql -U postgres << EOF
ALTER SYSTEM SET google_columnar_engine.memory_size_in_mb=30720;
ALTER SYSTEM SET
google_columnar_engine.relations='tpch.public.customer,tpch.public.lineitem,tpch.public.n
ation,tpch.public.orders,tpch.public.part,tpch.public.partsupp,tpch.public.region,tpch.pu
blic.supplier';
EOF
```

2. Connect to the **server** machine and restart the database so it picks up the new settings:

```
sudo alloydb database-server stop
sudo alloydb database-server start
```

3. Wait for the restart operation to finish. It can take up to a few minutes to complete.

4. Connect to the **client** machine and monitor the population of columnar-engine as follows:

   a. Confirm that `google_columnar_engine.enabled` is set to `on`. Use the command `psql -h $PGHOST -U postgres -c "SHOW google_columnar_engine.enabled"` for this purpose.

   b. Check the status of columnar engine population within the `tpch` database by using the following command.

   ```
   psql -U postgres -d tpch
   ...
   tpch=> select relation_name, block_count_in_cc, total_block_count,
   block_count_in_cc=total_block_count from g_columnar_relations order by 1; \watch
   10
   ```

   Note `\watch 10` at the end of the SQL command which executes the command every 10 seconds. You should observe the output of this command until it no longer changes. Once the output stops changing, specifically, check block_count_in_cc=total_block_count for every relation, go to the next step for validation of the output. Also as a general rule of thumb, ensure that the `total_block_count` matches `block_count_in_cc` for all the relations.

   c. Validate the status of columnar engine population as follows, in particular make sure the final column is "`t`" for all rows:

   <u>Validation State for Scale Factor = 10</u>

   Below is the final state of the columnar engine after population was done for scale factor 10:

```
psql -U postgres -d tpch

tpch=> select relation_name, block_count_in_cc, total_block_count,
block_count_in_cc=total_block_count from g_columnar_relations order by 1;

 relation_name | block_count_in_cc | total_block_count | ?column?
---------------+-------------------+-------------------+----------
 customer      |             36657 |             36657 | t
 lineitem      |           1331456 |           1331456 | t
 nation        |                 1 |                 1 | t
 orders        |            278724 |            278724 | t
 part          |             42612 |             42612 | t
 partsupp      |            184108 |            184108 | t
 region        |                 1 |                 1 | t
 supplier      |              2269 |              2269 | t
(8 rows)
```

Validation state for Scale Factor = 30

```
 relation_name | block_count_in_cc | total_block_count | ?column?
---------------+-------------------+-------------------+----------
 customer      |            110284 |            110284 | t
 lineitem      |           4025262 |           4025262 | t
 nation        |                 1 |                 1 | t
 orders        |            841603 |            841603 | t
 part          |            126433 |            126433 | t
 partsupp      |            551690 |            551690 | t
 region        |                 1 |                 1 | t
 supplier      |              6805 |              6805 | t
(8 rows)
```

Tuning for Scale Factor 100

The size of the `tpch` database that we load with `TPCH_SCALE=100` is approximately 205GB. This database size is substantially larger than the size of available RAM on the machine (128 GB) of type 16 virtual CPUs. We cannot therefore populate the columnar engine for the entire database. This is where AlloyDB Columnar Engine's `auto columnarization` comes into action. Note that we must let CE observe the workload first, and the tuning steps here differ slightly. After we enable the `Columnar Engine`, we need to execute the entire set of TPC-H queries once. That enables the recommendation engine to make suggestions on the optimal values to set for `google_columnar_engine.relations` and `google_columnar_engine.memory_size_in_mb` database flags.

Below are the simple tuning steps:

1. Connect to the **client** machine and run the following command to edit important Columnar Engine flags:

```
psql -U postgres << EOF
ALTER SYSTEM SET google_columnar_engine.memory_size_in_mb=40960;
EOF
```

2. Connect to the **server** machine and restart the database so it picks up the new settings:
   ```
   sudo alloydb database-server stop
   sudo alloydb database-server start
   ```

3. Wait for the restart operation to finish. It can take up to a few minutes to complete.

4. Confirm that `google_columnar_engine.enabled` is set to `on`. Use `psql -U postgres -c "SHOW google_columnar_engine.enabled"` command to confirm this.

5. Reset the columnar engine recommendation by using the following command:
   ```
   psql -U postgres -d tpch -c "SELECT google_columnar_engine_reset_recommendation('true')";
   ```

6. **`Observe workload`**: In this step, you simply execute all of the 22 TPC-H queries (just once) that will let Columnar Engine observe the workload to make optimal tuning suggestions. You can create and execute the following script to train the engine (execute it from `hammerdb/HammerDB-4.6` directory):

   Create the script `train-recommendation-engine.sh` with the following content:

   ```
   #!/bin/bash -x

   source ./setup.env

   ./hammerdbcli << EOF

   # CONFIGURE PARAMETERS FOR TPC-H BENCHMARK
   # --------------------------------------
   dbset db pg
   dbset bm tpc-h

   # CONFIGURE POSTGRES HOST AND PORT
   # --------------------------------------
   diset connection pg_host $PGHOST
   diset connection pg_port $PGPORT

   # CONFIGURE TPC-H
   # --------------------------------------
   diset tpch pg_tpch_superuser postgres
   diset tpch pg_tpch_user postgres
   diset tpch pg_tpch_dbase tpch
   diset tpch pg_scale_fact $TPCH_SCALE
   diset tpch pg_num_tpch_threads 1
   ```

```
diset tpch pg_degree_of_parallel 8
vuset vu 1

# logging
vuset logtotemp 1
vuset timestamps 0
vuset unique 0

# load tpc-h script and run benchmark
loadscript
vurun

# terminate when completed
waittocomplete
vudestroy
quit

EOF
```

Execute `train-recommendation-engine.sh` script as follows:

```
chmod +x ./train-recommendation-engine.sh
mkdir -p results
sudo nohup ./train-recommendation-engine.sh > results/train-recommendation-engine.out
2>&1
```

7. **Optimal tuning suggestion**: Once all the queries from previous step finish to execute, run the following command to find the optimal columnar engine tuning for `tpch` database:

```
psql -U postgres -d tpch  -c "SELECT google_columnar_engine_recommend('RECOMMEND_SIZE')"
```

   a. This command uses the recommendation engine to recommend the performance optimal memory size and recommended column.

   b. <u>**Output**</u> looks like following:

```
(39010,"tpch.public.customer(c_acctbal,c_address,c_comment,c_custkey,c_mktsegment,c
_name,c_nationkey,c_phone),tpch.public.lineitem(l_commitdate,l_discount,l_extendedp
rice,l_linestatus,l_orderkey,l_partkey,l_quantity,l_receiptdate,l_returnflag,l_ship
date,l_shipinstruct,l_shipmode,l_suppkey,l_tax),tpch.public.orders(o_custkey,o_orde
rdate,o_orderkey,o_orderpriority,o_shippriority),tpch.public.part(p_brand,p_contain
er,p_name,p_partkey,p_size,p_type),tpch.public.partsupp(ps_partkey,ps_suppkey,ps_su
pplycost),tpch.public.supplier(s_address,s_comment,s_name,s_nationkey,s_suppkey)")
```

c. Note the 2 parts of the above output:

   i. **First Part**: It is an integer (in this case, `39010`). This is the recommended value for the `google_columnar_engine.memory_size_in_mb` parameter. However, we can safely disregard this parameter as the difference between the new suggested value and the original value we specified (`40960`) is not significant.

   ii. **Second Part**: A string containing a list of recommended relations and their important columns `"tpch.public.customer(c_acctbal,c_address,c_comment,c_custkey,c_mktsegment,c_name,c_nationkey,c_phone),tpch.public.lineitem(l_commitdate,l_discount,l_extendedprice,l_linestatus,l_orderkey,l_partkey,l_quantity,l_receiptdate,l_returnflag,l_shipdate,l_shipinstruct,l_shipmode,l_suppkey,l_tax),tpch.public.orders(o_custkey,o_orderdate,o_orderkey,o_orderpriority,o_shippriority),tpch.public.part(p_brand,p_container,p_name,p_partkey,p_size,p_type),tpch.public.partsupp(ps_partkey,ps_suppkey,ps_supplycost),tpch.public.supplier(s_address,s_comment,s_name,s_nationkey,s_suppkey)"`.

8. Connect to AlloyDB Omni from the client using `psql -U postgres -d tpch` and run the following command to set `google_columnar_engine.relations = "<`**`Second Part`**`>"`.

```
ALTER SYSTEM SET
google_columnar_engine.relations='tpch.public.customer(c_acctbal,c_address,c_comment,c_cu
stkey,c_mktsegment,c_name,c_nationkey,c_phone),tpch.public.lineitem(l_commitdate,l_discou
nt,l_extendedprice,l_linestatus,l_orderkey,l_partkey,l_quantity,l_receiptdate,l_returnfla
g,l_shipdate,l_shipinstruct,l_shipmode,l_suppkey,l_tax),tpch.public.orders(o_custkey,o_or
derdate,o_orderkey,o_orderpriority,o_shippriority),tpch.public.part(p_brand,p_container,p
_name,p_partkey,p_size,p_type),tpch.public.partsupp(ps_partkey,ps_suppkey,ps_supplycost),
tpch.public.supplier(s_address,s_comment,s_name,s_nationkey,s_suppkey)';
```

   Note: The strings in Postgres/AlloyDB Omni use single-quotes, **not** double-quotes.

9. Connect to the server VM and restart the database so it picks up the new settings:

```
sudo alloydb database-server stop
sudo alloydb database-server start
```

10. Run the query `SELECT * from g_columnar_relations` regularly, and wait until values do not change any further. Use the following SQL command with `\watch 10` switch to allow the query to execute in every 10 seconds.

```
psql -U postgres -d tpch
...
tpch=> select relation_name, block_count_in_cc, total_block_count,
block_count_in_cc=total_block_count from g_columnar_relations order by 1; \watch 10
```

Observe the output of this command until it no longer changes. Once the output stops changing, ensure that the `total_block_count` matches `block_count_in_cc` for all the relations. The final state of `g_columnar_relations` should be close to the following:

```
 relation_name | total_block_count | block_count_in_cc | ?column?
---------------+-------------------+-------------------+----------
 customer      |            366517 |            366517 | t
 lineitem      |          13497033 |          13497033 | t
 orders        |           2816940 |           2816940 | t
 part          |            419467 |            419467 | t
 partsupp      |           1840017 |           1840017 | t
 supplier      |             22686 |             22686 | t
(6 rows)
```

Running the TPC-H benchmark

In this stage, we perform the TPC-H benchmark's "**Power Test**" with one client running 22 TPC-H queries, monitoring each query's response time, and calculating a final "**Geometric mean of query times returning rows**."

What is Power Test in TPROC-H?

The TPROC-H "**Power Test**" in HammerDB is a performance test that measures the ability of a database system to handle large-scale data warehousing workloads. HammerDB utilizes a modified version of TPC-H "power test" that does not have refresh functions. In this test, a single client generates a series of 22 queries that simulate typical data warehousing operations, such as generating reports, analyzing data, and performing complex joins. The test is based on the TPC-H benchmark, which is a standard benchmark used to evaluate the performance of database systems for data warehousing applications. The goal of the TPROC-H Power test is to measure the minimum query latency (or response time) that can be achieved by a single client, which provides an indication of the overall performance and scalability of the database system under test.

What is the "Geometric Mean" Metric?

The geometric mean is a measure of central tendency that is used in the TPROC-H benchmark. It is calculated by taking the product of all of the query times and then taking the n-th root of the product, where *n* is the number of queries. The geometric mean is used in the TPROC-H benchmark because it is less sensitive to outliers than the arithmetic mean. The arithmetic mean is the average of all of the query times. However, if there is one query that takes a very long time, the arithmetic mean will be skewed by that query. The geometric mean, on the other hand, is not as sensitive to such outliers. Even if one query takes a very long time, the product of all of the query times will not be as affected by that query. Refer to TPC-H official documentation to learn more about this metric.

A lower geometric mean of query times returning rows is desirable, as it indicates that the database system can process queries more quickly and efficiently and can handle larger data volumes more effectively.

Use the following script to execute the "**Power Test**" benchmark for TPROC-H. This script repeats the series of 22 queries. The first set of the query executions is intended to warm up the database caches, while the second set is used for actual performance measurement.

1. Switch to benchmark home directory:

```
cd hammerdb/HammerDB-4.6

source ./setup.env
```

2. Create `run-tpch.sh` script as follows:

```
#!/bin/bash -x

source ./setup.env

./hammerdbcli << EOF

# CONFIGURE PARAMETERS FOR TPC-H BENCHMARK
# --------------------------------------
dbset db pg
dbset bm tpc-h

# CONFIGURE POSTGRES HOST AND PORT
# --------------------------------------
diset connection pg_host $PGHOST
diset connection pg_port $PGPORT

# CONFIGURE TPC-H
# --------------------------------------
diset tpch pg_tpch_superuser postgres
diset tpch pg_tpch_user postgres
diset tpch pg_tpch_dbase tpch
diset tpch pg_scale_fact $TPCH_SCALE
diset tpch pg_num_tpch_threads 1
diset tpch pg_refresh_on false
diset tpch pg_refresh_verbose false
diset tpch pg_degree_of_parallel 8
diset tpch pg_trickle_refresh 1000
diset tpch pg_tpch_tspace pg_default
diset tpch pg_tpch_gpcompat false
diset tpch pg_tpch_gpcompress false
diset tpch pg_cloud_query false
diset tpch pg_rs_compat false
diset tpch pg_update_sets 1
diset tpch pg_total_querysets 1
```

```
vuset vu 1

# logging
vuset logtotemp 1
vuset timestamps 0
vuset unique 0

# load tpc-h script and run benchmark
loadscript
# Warmup run
vurun

# Measurement run
vurun

# terminate when completed
waittocomplete
vudestroy
quit

EOF
```

3. Run the script as follows:

```
chmod +x run-tpch.sh
mkdir -p results
sudo nohup ./run-tpch.sh > results/run-tpch.out 2>&1
```

4. Below is a sample output of `run-tpch.sh` script obtained for scenario where Columnar-Engine (CE) is enabled and TPC-H `scale factor` is set to `30`:

```
TPROC-H Driver Script
Script loaded, Type "print script" to view


Vuser 1 created - WAIT IDLE
Failed to create virtual users: Could not open tempfile /tmp/hammerdb.log
Vuser 1:RUNNING
Vuser 1:Executing Query 14 (1 of 22)
Vuser 1:query 14 completed in 9.569 seconds
Vuser 1:Executing Query 2 (2 of 22)
Vuser 1:query 2 completed in 18.363 seconds
Vuser 1:Executing Query 9 (3 of 22)
…
…
Vuser 1:query 12 completed in 4.194 seconds
Vuser 1:Completed 1 query set(s) in 314 seconds
Vuser 1:Geometric mean of query times returning rows (22) is 7.37605
Vuser 1:FINISHED SUCCESS
ALL VIRTUAL USERS COMPLETE
TPROC-H Driver Script
jobid=642777185F8303E203936333

Vuser 1:RUNNING
Vuser 1:Executing Query 14 (1 of 22)
Vuser 1:query 14 completed in 3.086 seconds
Vuser 1:Executing Query 2 (2 of 22)
Vuser 1:query 2 completed in 14.241 seconds
Vuser 1:Executing Query 9 (3 of 22)
…
…
Vuser 1:query 12 completed in 4.271 seconds
Vuser 1:Completed 1 query set(s) in 264 seconds
Vuser 1:Geometric mean of query times returning rows (22) is 6.05468
Vuser 1:FINISHED SUCCESS
ALL VIRTUAL USERS COMPLETE
TPROC-H Driver Script
jobid=642778545F8303E273233383
```

NOTE: As stated previously, we only evaluate the second round of query execution when measuring performance. The initial round serves as a warm-up.

Expected TPC-H Results

The table below summarizes the execution time (in seconds) for each of the 22 TPC-H queries. As stated previously, three distinct scenarios with scale factors of 10, 30, and 100 have been explored. In each scenario, the query execution durations and geometric mean for all queries with Columnar-Engine (CE) population are presented. You should anticipate TPC-H (power test) performance results similar to the following:

Query Id	**Query Execution Time (in seconds)**		
	TPCH_SCALE **= 10**	TPCH_SCALE **= 30**	TPCH_SCALE **= 100**
1	3.968	13.021	40.21
2	2.514	16.964	60.591
3	1.883	5.398	18.325
4	0.452	1.287	32.819
5	1.527	3.476	7.441
6	0.208	0.132	0.371
7	2.03	3.163	17.243
8	0.642	1.982	561.564

9	4.371	15.826	1510.82
10	1.877	5.543	16.106
11	0.833	2.836	102.489
12	0.546	1.658	5.491
13	5.003	13.971	105.958
14	0.378	1.175	2.581
15	2.96	5.813	20.314
16	1.412	3.921	21.894
17	6.211	22.293	229.619
18	17.704	55.445	263.422
19	0.083	0.258	0.64
20	1.207	23.764	8281.706
21	1.636	6.302	977.062
22	0.186	0.345	0.739
Geometric mean (seconds)	**1.30**	**3.92**	**35.8**

OLAP Atomics Benchmarking

To evaluate and improve the OLAP capabilities of AlloyDB's Columnar-Engine, the engineers at Google have developed a custom benchmark known as **OLAP atomics**, which consists of a collection of 11 primitive queries executed over a large volume of data and covering the fundamental operations of an OLAP system. This set of primitive OLAP queries can perform the fundamental data manipulation and analysis of any typical OLAP system, including selection, slice-and-dice, joins, roll-up (also known as aggregation or consolidation), drill-down, etc.

Measuring OLAP atomics on a database system is important because it can reveal performance bottlenecks at the primitive level. These primitive queries are a valuable tool for ensuring that the OLAP system fulfills the requirements for your large-scale data analysis and decision support.

For the purposes of this user guide, a TPC-H database with a **scale factor of 30** was utilized.

Setup, Configuration and Tuning

Before you can execute the OLAP atomic queries, you must perform the database configuration and tuning described in this section.

Prerequisites

A. You need to run the following steps from a client (driver) machine. Ensure that you have completed the setup steps listed in the "Setup of Benchmark Driver Machine (Client)" section (especially installation of the HammerDB utility).

B. **Cleanup**: If you are running multiple benchmarks in succession, ensure you follow the "Benchmark Cleanup" section before doing your subsequent run.

Initial Setup on Client Machine

Connect to the client machine and execute the following commands:

```
cd hammerdb/HammerDB-4.6
```

Then create `setup.env` file as follows:

```
cat << EOF > setup.env

# Private IP of the AlloyDB primary instance
export PGHOST=111.222.333.444

# Postgres default port address. You do not need to change it unless you use non-default port
address.
export PGPORT=5432   # default port to connect with postgres

# TPC-H Scale Factor (determines the size of the database that we want to build).
export TPCH_SCALE=30

EOF
```

Edit the above file and all the settings (excluding `TPCH_SCALE`, that should remain as 30) to suit your environment.

Now, to load the TPC-H database, follow the exact steps outlined in the Script to load TPC-H data section. The load steps are identical to the TPC-H benchmarking.

Altering the TPC-H schema

For the purpose of OLAP atomics, we only need the `lineitem` and `supplier` tables from `tpch` database (i.e. without any constraints or indices). In this section, we provide minimal instructions to prepare the database for query execution.

1. Connect to the client machine.

2. Connect to the `tpch` database by using `psql -h $PGHOST -U postgres -d tpch` command.

3. Now run the following commands to drop all the constraints and indices from lineitem and supplier tables:

```
--- Drop constraints from lineitem table:
ALTER TABLE lineitem DROP CONSTRAINT IF EXISTS lineitem_pk CASCADE;
ALTER TABLE lineitem DROP CONSTRAINT IF EXISTS lineitem_partsupp_fk CASCADE;
ALTER TABLE lineitem DROP CONSTRAINT IF EXISTS lineitem_order_fk CASCADE;

--- Drop all indexes of lineitem table:
DROP INDEX IF EXISTS lineitem_part_supp_fkidx CASCADE;
DROP INDEX IF EXISTS idx_lineitem_orderkey_fkidx CASCADE;
DROP INDEX IF EXISTS lineitem_pk CASCADE;

--- Drop constraints from supplier table:

ALTER TABLE supplier DROP CONSTRAINT IF EXISTS supplier_pk CASCADE;
ALTER TABLE supplier DROP CONSTRAINT IF EXISTS supplier_nation_fk CASCADE;
ALTER TABLE supplier DROP CONSTRAINT IF EXISTS "2200_127555_1_not_null" CASCADE;

--- Drop all indexes of supplier table:
DROP INDEX IF EXISTS supplier_nation_fkidx CASCADE;

--- Drop all the tables that are not needed:
DROP TABLE customer CASCADE;
DROP TABLE nation CASCADE;
DROP TABLE orders CASCADE;
DROP TABLE part CASCADE;
DROP TABLE partsupp CASCADE;
DROP TABLE region CASCADE;
```

4. Verify that you only see the following objects in the tpch database after executing the preceding commands:

```
tpch=> \dti+
                               List of relations
 Schema |   Name   | Type  |  Owner   | Table | Persistence | Size  | Description
--------+----------+-------+----------+-------+-------------+-------+-------------
 public | lineitem | table | postgres |       | permanent   | 31 GB |
 public | supplier | table | postgres |       | permanent   | 53 MB |
(2 rows)
```

Tuning Columnar Engine

Here are the recommended procedures for tuning the AlloyDB columnar engine:

1. Connect to the **client** machine and run the following commands to edit important Columnar Engine flags:

```
psql -U postgres << EOF
ALTER SYSTEM SET google_columnar_engine.enabled=on;
ALTER SYSTEM SET google_columnar_engine.memory_size_in_mb=39322;
ALTER SYSTEM SET max_parallel_workers_per_gather=2;
ALTER SYSTEM SET max_parallel_workers=16;
EOF
```

2. Connect to the **server** machine and restart the database so it picks up the new settings

```
sudo alloydb database-server stop
sudo alloydb database-server start
```

3. Wait for the restart operation to finish. It can take up to a few minutes to complete.

4. Connect to the client machine and verify that the Columnar Engine configurations are in effect. Use following command to confirm this:

```
psql -U postgres << EOF
SHOW google_columnar_engine.enabled;
SHOW google_columnar_engine.memory_size_in_mb;
SHOW max_parallel_workers_per_gather;
SHOW max_parallel_workers;
EOF
```

Verify that the output matches the following:

```
 google_columnar_engine.enabled
--------------------------------
 on
(1 row)

 google_columnar_engine.memory_size_in_mb
------------------------------------------
 39322
(1 row)

 max_parallel_workers_per_gather
---------------------------------
 2
(1 row)

 max_parallel_workers
----------------------
 16
(1 row)
```

5. Connect to the `tpch` database by using `psql -h $PGHOST -U postgres -d tpch` command and then run the following commands to add `lineitem` and `supplier` tables to the columnar-engine.

```
SELECT google_columnar_engine_add('lineitem');
SELECT google_columnar_engine_add('supplier');
```

6. Validation of columnar-engine population: Use the command `psql -h $PGHOST -U postgres -d tpch` and then run the query `SELECT relation_name, block_count_in_cc, total_block_count, block_count_in_cc=total_block_count from g_columnar_relations order by 1; \watch 10` and ensure that the output reaches a comparable state to the following:

```
relation_name | block_count_in_cc | total_block_count | ?column?
---------------+-------------------+-------------------+----------
 lineitem      |           4024945 |           4024945 | t
 supplier      |              6805 |              6805 | t
(2 rows)
```

Now we are ready to execute the OLAP atomics benchmark.

Queries in OLAP Atomics

The following table summarizes the customized OLAP queries that are executed on the `tpch` database that we just loaded. The engineering team at Google AlloyDB develops these queries.

Scenario Description	**Query**	**Query Id**
Aggregation (count operation) with a filter covering approximately 10% of the large lineitem table.	`select count(l_orderkey) from lineitem where l_discount = 0;`	Q1
Aggregation (SUM) on an integer column with a filter covering approximately 10% of the lineitem table.	`select sum(l_linenumber) from lineitem where l_discount = 0;`	Q2
Aggregation (SUM) on numeric column with a filter covering approximately 10% of the lineitem table.	`select sum(l_quantity) from lineitem where l_discount = 0;`	Q3
Summarization using GROUP BY and AGGREGATION on the entire lineitem table.	`select count(l_shipmode), l_shipmode from lineitem group by l_shipmode;`	Q4
Full table scan without any filters	`select count(l_comment) from lineitem;`	Q5
Full table scan with equality predicate (filter)	`select count(*) from lineitem where l_quantity=25.99;`	Q6
Sorting of the entire table and presenting the top values	`select l_orderkey, l_commitdate, l_shipmode from lineitem order by 1,2,3 limit 10;`	Q7
Full table scan with LIKE predicate	`select count(*) from lineitem where l_shipinstruct like '%DE%';`	Q8
LIST based selection on the entire table	`select count(*) from lineitem where l_tax in (0.01, 0.02, 0.05);`	Q9

MIX and MAX aggregation on the entire table	`select min(l_quantity), max(l_discount) from lineitem;`	Q10
Join with a predicate	`select count(*) from supplier, lineitem where s_acctbal = l_extendedprice;`	Q11

Execute OLAP Atomics

The execution of OLAP atomic queries is as simple as connecting to the `tpch` database and executing the queries introduced in section [Queries in OLAP Atomics](#).

It is **recommended** to execute the queries using "**EXPLAIN ANALYZE** <query> .." prefix clause, which will display the query plan and execution time.

Below is an example of executing Q1 from `tpch` database:

```
tpch=> explain analyze select count(l_orderkey) from lineitem where l_discount = 0;

QUERY PLAN
------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------
 Finalize Aggregate  (cost=137113.77..137113.78 rows=1 width=8) (actual time=151.708..153.953
rows=1 loops=1)
   ->  Gather  (cost=137113.56..137113.77 rows=2 width=8) (actual time=151.693..153.944 rows=3
loops=1)
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=136113.56..136113.57 rows=1 width=8) (actual
time=145.574..145.576 rows=1 loops=3)
               ->  Parallel Append  (cost=20.00..119102.90 rows=6804263 width=7) (actual
time=0.063..145.569 rows=5454623 loops=3)
                     ->  Parallel Custom Scan (columnar scan) on lineitem
(cost=20.00..119098.89 rows=6804262 width=7) (actual time=0.062..145.565 rows=5454623 loops=3)
                           Filter: (l_discount = '0'::numeric)
                           Rows Removed by Columnar Filter: 54546385
                           Rows Aggregated by Columnar Scan: 1904168
                           Columnar cache search mode: native
                     ->  Parallel Seq Scan on lineitem  (cost=0.00..4.01 rows=1 width=7) (never
executed)
                           Filter: (l_discount = '0'::numeric)

 Planning Time: 7.75 ms
 Execution Time: 159.008 ms
(15 rows)
```

You should note the `Execution Time` in the above output, which is significantly faster for `AlloyDB` columnar-engine.

Expected Results

The following table gives a summary of the queries to execute and their expected execution and planning time.

Query Id	**Query To Execute**	Execution Time (milliseconds)	Planning Time (ms)
Q1	`EXPLAIN ANALYZE SELECT COUNT(l_orderkey) FROM lineitem WHERE l_discount = 0;`	159.00	7.75
Q2	`EXPLAIN ANALYZE SELECT SUM(l_linenumber) FROM lineitem WHERE l_discount = 0;`	279.00	2.25
Q3	`EXPLAIN ANALYZE SELECT SUM(l_quantity) FROM lineitem WHERE l_discount = 0;`	278.00	2.03
Q4	`EXPLAIN ANALYZE SELECT COUNT(l_shipmode), l_shipmode FROM lineitem GROUP BY l_shipmode;`	912.00	1.86
Q5	`EXPLAIN ANALYZE SELECT COUNT(l_comment) FROM lineitem;`	218.00	1.96
Q6	`EXPLAIN ANALYZE SELECT COUNT(*) FROM lineitem WHERE l_quantity=25.99;`	1.61	2.30
Q7	`EXPLAIN ANALYZE SELECT l_orderkey, l_commitdate, l_shipmode FROM lineitem ORDER BY 1,2,3 LIMIT 10;`	2062.00	2.09
Q8	`EXPLAIN ANALYZE SELECT COUNT(*) FROM lineitem WHERE l_shipinstruct like '%DE%';`	286.00	2.23
Q9	`EXPLAIN ANALYZE SELECT COUNT(*) FROM lineitem WHERE l_tax in (0.01, 0.02, 0.05);`	368.00	2.38
Q10	`EXPLAIN ANALYZE SELECT MIN(l_quantity), MAX(l_disCOUNT) FROM lineitem;`	378.00	5.55
Q11	`EXPLAIN ANALYZE SELECT COUNT(*) FROM supplier, lineitem WHERE s_acctbal = l_extendedprice;`	6,774.00	1.95